Лабораторная работа №3. Управление процессами в ОС Linux

Рассматриваемые вопросы

1. Директивы объединения команд
2. Команды для управления процессами
3. Планирование времени запуска процессов
4. Организация межпроцессного взаимодействия

Директивы (команды) объединения команд

Командный интерпретатор **bash** поддерживает следующие директивы объединения команд:

команда1 | команда2 – перенаправление стандартного вывода,

команда1 ; команда2 – последовательное выполнение команд,

команда1 && команда2 – выполнение команды при успешном завершении предыдущей,

команда1 || команда2 – выполнение команды при неудачном завершении предыдущей,

команда1 $(команда2) – передача результатов работы команды 2 в качестве аргументов запуска команды 1,

команда 1 > файл– направление стандартного вывода в файл (содержимое существующего файла удаляется),

команда 1 >> файл – направление стандартного вывода в файл (поток дописывается в конец файла).

```
{
команда1
команда 2
}
```
– объединение команд после директив **||** , **&&** или в теле циклов и функций.

команда1 & – запуск команды в фоновом режиме (стандартный вход и стандартный выход не связаны с консолью, из которой запускается процесс; управление процессом возможно в общем случае только с помощью сигналов).

Команды для управления процессами

(с подробным описанием возможностей и синтаксисом команд можно ознакомиться в документации, доступной по команде **man команда**)

kill – передает сигнал процессу. Сигнал может передаваться в виде его номера или символьного обозначения. По умолчанию (без указания сигнала) передает сигнал требования о завершении процесса завершения процесса (**TERM**). Идентификация процесса для команды **kill** производится по PID. Перечень системных сигналов, доступных в GNU/Linux, с указанием их номеров и символьных обозначений можно получить с помощью команды **kill –l**;

killall – работает аналогично команде kill, но для идентификации процесса использует его символьное имя, а не PID;

pidof – определяет PID процесса по его имени;

pgrep – определяет PID процессов с заданными характеристиками (например, запущенные конкретным пользователем);

pkill – позволяет отправить сигнал группе процессов с заданными характеристиками;

nice – запускает процесс с заданным значением приоритета. Уменьшение значения (повышение приоритета выполнения) может быть инициировано только пользователем root;

renice – изменяет значения приоритета для запущенного процесса. Уменьшение значения (повышение приоритета выполнения) может быть инициировано только пользователем root;

at – осуществляет однократный отсроченный запуск команды.

cron – демон, который занимается планированием и выполнением команд, запускаемых по определенным датам и в определенное время. Команды, выполняемые периодически, указываются с использованием команды **crontab**. Команды, которые должны быть запущены лишь однажды, добавляются при помощи **at.** Синтаксис строки в **crontab** подробно описан здесь: http://www.opennet.ru/man.shtml?topic=crontab&category=5&russian=2.

tail – не только выводит последние n строк из файла, но и позволяет организовать "слежение" за файлом – обнаруживать и выводить новые строки, появляющиеся в конце файла.

sleep – задает паузу в выполнении скрипта.

Организация взаимодействия двух процессов

Существует несколько вариантов организации взаимодействия процессов. Поскольку суть взаимодействия состоит в передаче данных и/или управления от одного процесса к другому, рассмотрим два распространенных варианта организации такого взаимодействия: передачу данных через именованный канал и передачу управления через сигнал.

Взаимодействие процессов через именованный канал

Именованный канал – специальный тип файла в Linux. Создается командой **mkfifo имя_файла**.

Взаимодействие с именованным каналом происходит обычными средствами для взаимодействия с файлами, но

при этом такой файл не будет сохраняться на носителе, а представляет собой буфер в памяти для организации межпроцессного обмена данными.
Для демонстрации передачи информации через именованный канал рассмотрим два скрипта – «Генератор» и «Обработчик». Требуется считывать информацию с одной консоли с помощью процесса «Генератор» и выводить ее на экран другой консоли с помощью процесса «Обработчик», причем таким образом, чтобы считывание генератором строки «QUIT» приводило к завершению работы обработчика. Каждый скрипт запускается независимо в своей виртуальной консоли. Переключаясь между консолями, можно управлять скриптами и наблюдать результаты их работы.
Перед запуском скриптов создадим именованный канал с помощью команды **mkfifo pipe**

Генератор	Обработчик
#!/bin/bash while true; do read LINE echo $LINE > pipe done	#!/bin/bash (tail -f pipe) | while true; do read LINE; case $LINE in QUIT) echo "exit" killall tail exit ;; *) echo $LINE ;; esac done

Скрипт «Генератор» в бесконечном цикле считывает строки с консоли и записывает их в именованный канал **pipe**.
Скрипт «Обработчик» рассмотрим подробнее.
Команда **tail** позволяет считывать последние **n** строк из файла. Но один из наиболее распространенных вариантов ее использования – организация «слежения» за файлом. При использовании конструкции **tail -f** считывание из файла будет происходить только в случае добавления информации в этот файл. Поскольку необходимо передавать выход команды **tail** на вход скрипта «Обработчик», используем конструкцию **(команды) |** Круглые скобки позволяют запустить независимый подпроцесс (дочерний процесс) внутри родительского процесса «Обработчик», а оператор конвейера в конце позволит направить выход этого подпроцесса на вход родительского процесса. Таким образом, команда **read** в этом скрипте читает выход команды **tail.** Остальная часть скрипта основывается на конструкциях, изученных в предыдущих лабораторных работах, и не требует детального рассмотрения. Исключение составляет только команда **killall tail.** С ее помощью завершается вызванный в подпроцессе процесс **tail** перед завершением родительского процесса. Использование **killall** в этом случае используется для упрощения кода, но не всегда является корректным. Лучше определять PID конкретного процесса **tail,** вызванного в скрипте, и завершать его с помощью команды **kill.**

Взаимодействие процессов с помощью сигналов
Сигналы являются основной формой передачи управления от одного процесса к другому. В Linux используются 64 различных сигнала. Любой процесс при создании наследует от родительского процесса таблицу указателей на обработчики сигналов и, тем самым, готов принять любой из 64-х сигналов. Обработчики по-умолчанию, как правило, приводят к завершению процесса или, реже, игнорированию сигнала, но за исключение двух сигналов (**KILL** и **STOP**), обработчики могут быть заменены на другие. У большинства сигналов есть предопределенное назначение и ситуации, в которых они будут передаваться, поэтому изменение их обработчиков возможно, но рекомендуется только в рамках расширения функционала соответствующего обработчика. Такие сигналы считаются системными. Но есть два сигнала (**USR1** и **USR2**), для которых предполагается, что обработчики будут создаваться разработчиком приложения и использоваться для передачи и обработки пользовательских, неспецифицированных сигналов (по умолчанию, их обработка предполагает завершение процесса). Для замены обработчиков сигналов в **sh** (**bash**) используется встроенная команда **trap** с форматом

trap action signal

Команде нужно передать два параметра: действие при получении сигнала и сигнал, для которого будет выполняться указанное действие. Обычно в качестве действия указывают вызов функции, описанной выше в коде скрипта.

С помощью команды **trap** можно не только задать обработчик для пользовательского сигнала, но и подменить обработчик для некоторых из системных сигналов (кроме тех, перехват которых запрещен). В этом случае обработка сигнала перейдет к указанному в **trap** обработчику.

Для демонстрации передачи управления от одного процесса к другому рассмотрим еще одну пару скриптов.

Генератор	Обработчик

#!/bin/bash
while true; do
read LINE
case $LINE in
STOP)
kill -USR1 $(cat .pid)
;;
*)
:
;;
esac
done
	#!/bin/bash
	echo $$ > .pid
	A=1
	MODE="rabota"
	usr1()
	{
	MODE="ostanov"
	}
	trap 'usr1' USR1
	while true; do
	case $MODE in
	"rabota")
	let A=$A+1
	echo $A
	;;
	"ostanov")
	echo "Stopped by SIGUSR1"
	exit
	;;
	esac
	sleep 1
	done

В этом случае скрипт «Генератор» будет в бесконечном цикле считывать строки с консоли и бездействовать (используется оператор **:**) для любой входной строки, кроме строки STOP, получив которую, он отправит пользовательский сигнал **USR1** процессу «Обработчик». Поскольку процесс «Генератор» должен знать PID процесса «Обработчик», передача этого идентификационного номера осуществляется через скрытый файл. В процессе «Обработчик» определение PID процесса производится с помощью системной переменной **$$.** Процесс «Обработчик» выводит на экран последовательность натуральных чисел до момента получения сигнала **USR1**. В этот момент запускается обработчик **usr1()**, который меняет значение переменной MODE. В результате на следующем шаге цикла будет выведено сообщение о прекращении работы в связи с появлением сигнала, и работа скрипта будет завершена.

Задание на лабораторную работу

Создайте скрипты или запишите последовательности выполнения команд для перечисленных заданий и предъявите их преподавателю.

1. Создайте и однократно выполните скрипт (в этом скрипте нельзя использовать условный оператор и операторы проверки свойств и значений), который будет пытаться создать директорию **test** в домашней директории. Если создание директории пройдет успешно, скрипт выведет в файл **~/report** сообщение вида "**catalog test was created successfully**" и создаст в директории **test** файл с именем **Дата_Время_Запуска_Скрипта**. Затем независимо от результатов предыдущего шага скрипт должен опросить с помощью команды **ping** хост www.net_nikogo.ru и, если этот хост недоступен, дописать сообщение об ошибке в файл **~/report.** Сообщение об ошибке должно начинаться с текущей **Дата_Время**, а затем содержать через пробел произвольный текст сообщения об ошибке.

2. Задайте еще один однократный запуск скрипта из пункта 1 через 2 минуты. Консоль после этого должна оставаться свободной. Выполнив отдельную команду организуйте слежение за файлом **~/report** и выведите на консоль новые строки из этого файла, как только они появятся.
3. Задайте запуск скрипта из пункта 1 в каждую пятую минут каждого часа в день недели, в который вы будете выполнять работу.
4. Создайте три фоновых процесса, выполняющих одинаковый бесконечный цикл вычисления (например, перемножение двух чисел). После запуска процессов должна сохраниться возможность использовать виртуальную консоль, с которой их запустили. Используя команду **top**, проанализируйте процент использования ресурсов процессора этими процессами. Создайте скрипт, который будет в автоматическом режиме обеспечивать, чтобы тот процесс, который был запущен первым, использовал ресурс процессора не более чем на 10%. Послав сигнал, завершите работу процесса, запущенного третьим. Проверьте, что созданный скрипт по-прежнему удерживает потребление ресурсов процессора первым процессом в заданном диапазоне.
5. Создайте пару скриптов: генератор и обработчик. Процесс «Генератор» передает информацию процессу «Обработчик» с помощью именованного канала. Процесс «Обработчик» должен осуществлять следующую обработку переданных строк: если строка содержит единственный символ «**+**», то процесс обработчик переключает режим на «сложение» и ждет ввода численных данных. Если строка содержит единственный символ «*****», то обработчик переключает режим на «умножение» и ждет ввода численных данных. Если строка содержит целое число, то обработчик осуществляет текущую активную операцию (выбранный режим) над текущим значением вычисляемой переменной и считанным значением (например, складывает или перемножает результат предыдущего вычисления со считанным числом). При запуске скрипта режим устанавливается в «сложение», а вычисляемая переменная приравнивается к 1. В случае получения строки **QUIT** скрипт «Обработчик» выдает сообщение о плановой остановке и оба скрипта завершают работу. В случае получения любых других значений строки оба скрипта завершают работу с сообщением об ошибке входных данных.
6. Создайте пару скриптов: генератор и обработчик. Процесс «Генератор» считывает с консоли строки в бесконечном цикле. Если считанная строка содержит единственный символ «**+**», он посылает процессу «Обработчик» сигнал **USR1**. Если строка содержит единственный символ «*****», генератор посылает обработчику сигнал **USR2**. Если строка содержит слово **TERM**, генератор посылает обработчику сигнал **SIGTERM** и завершает свою работу. Другие значения входных строк игнорируются. Обработчик добавляет 2 или умножает на 2 текущее значение обрабатываемого числа (начальное значение принять на единицу) в зависимости от полученного пользовательского сигнала и выводит результат на экран. Вычисление и вывод производятся один раз в секунду. Получив сигнал **SIGTERM**, «Обработчик» завершает свою работу, выведя сообщения о завершении работы по сигналу от другого процесса.